Universidade Federal do Rio Grande do Norte
Aluno: João Victor Andrade da Cunha Nascimento
Data: 12/10/2025
Professores: ANDRÉ MAURÍCIO CUNHA CAMPOS e RAFAEL BESERRA GOMES

Relatório do Compressor de Huffman

A análise de complexidade começa com a definição das variáveis que servirão como base para o estudo de desempenho dos programas envolvidos neste projeto:

- N — número total de símbolos (caracteres ou palavras-chave) no arquivo original, proporcional ao tamanho do arquivo.
- S — número de símbolos únicos presentes na tabela de frequência, como, por exemplo, {a, b, c, int, for}, onde S = 5.
- C — número total de bits que compõem o arquivo comprimido, diretamente proporcional ao tamanho do arquivo resultante.

Essas variáveis permitem expressar o custo computacional de cada etapa de forma quantitativa e comparável.. É importante fazermos está divisão pois mesmo em arquivos grandes, N grande, pode ser que o número de símbolos seja pequeno, S pequeno, se tornando uma nuance a ser analisada no relatório. O mesmo é válido para C que tem valor ativo durante a descompressão.

O programa contador tem uma operação central: contabilizar a frequência dos símbolos encontrados no arquivo. Esse processo é dividido em duas etapas principais:

Na primeira etapa, ocorre a leitura e tokenização do arquivo. O programa percorre todo o conteúdo de entrada, caractere por caractere, verificando se cada elemento faz parte de uma palavra, é um delimitador ou outro tipo de símbolo. Essa varredura é linear em relação ao tamanho do arquivo, resultando em uma complexidade de

→ O(N)

Na segunda etapa, ocorre o armazenamento dos dados na tabela de frequência. Para cada símbolo identificado, é feita uma atualização no std::map,

cuja estrutura é uma árvore binária balanceada. Cada inserção ou acesso nessa estrutura possui custo proporcional ao logaritmo do número de elementos armazenados. Assim, o custo total dessa etapa é

→ O(N log S)

Dessa forma, como a etapa de atualização do mapa é a que mais impacta no desempenho geral, a complexidade total do programa contador é dada por

→ O(N log S)

O programa compressor opera em dois modos distintos: compressão (-c) e descompressão (-d)..

O processo de compressão é composto por três fases principais:

Na construção da árvore de Huffman, a tabela de frequências é usada para gerar a estrutura hierárquica de códigos. Isso é feito com auxílio de uma fila de prioridade (min-heap), em que cada inserção possui um custo logarítmico, e as combinações são realizadas repetidamente para todos os símbolos. O custo total é

→ O(S log S)

A segunda fase consiste na geração dos códigos de Huffman, em que a árvore é percorrida integralmente para associar um código binário a cada símbolo. Esse percurso é linear em relação ao número de nós da árvore, com custo de

→ O(S)

Por fim, na codificação do arquivo original, o programa lê cada um dos símbolos e busca seu código correspondente no mapa de códigos. Como essas buscas ocorrem em uma estrutura do tipo mapa, cada operação tem custo logarítmico. O custo total dessa etapa é

→ O(N log S)

Com a soma das três fases, temos o custo geral:

→ O(S log S) + O(S) + O(N log S)

Considerando que, em geral, N é muito maior que S, o termo dominante será o de codificação do arquivo. Assim, a complexidade total de compressão pode ser expressa como

→ O(N log S)

A descompressão é mais direta e eficiente. A primeira fase é a leitura do arquivo comprimido, que contém C bits a serem carregados e processados. Esse custo cresce linearmente com o tamanho do arquivo, resultando em

→ O(C)

Em seguida, ocorre a decodificação por meio da árvore de Huffman, em que cada bit orienta uma única decisão (seguir à esquerda ou à direita). Essa operação é constante para cada bit, e como a sequência total tem C bits, a complexidade total é

→ O(C)

Portanto, a complexidade final do processo de descompressão é linear em relação ao tamanho do arquivo comprimido:

→ O(C)

Para avaliar a eficácia do algoritmo de Huffman implementado, foi conduzido um experimento comparativo. Foram selecionados dois conjuntos de arquivos: arquivos de texto puro (.txt) e arquivos de código-fonte em C++ (.cpp), representando o caso de uso geral e o caso de uso especializado, respectivamente.

Cada arquivo foi comprimido utilizando dois métodos:

Programa Próprio: O compressor desenvolvido neste projeto, que utiliza uma tabela de frequência fixa, pré-calculada a partir de uma análise de múltiplos arquivos de código C++.

Compress2Go (ZIP): Uma ferramenta online que oferece múltiplos formatos de compressão. Para este experimento, foi utilizada a opção de compressão para o formato ZIP, que geralmente implementa o algoritmo Deflate.

A eficiência de cada método foi medida pela taxa de compressão, calculada pela fórmula:

$$\text{Taxa de Compressão} = \left(1 - \frac{\text{Tamanho Comprimido}}{\text{Tamanho Original}}\right) \times 100\%$$

Os resultados obtidos nos testes foram compilados na tabela abaixo. Todos os tamanhos foram convertidos para bytes para garantir uma comparação precisa.

Tipo de Arquivo	Teste	Tamanho Original	Tamanho Comprimido (Meu Programa)	Tamanho Comprimido (Compress2Go)	Taxa de Compressão (Meu Programa)	Taxa de Compressão (Compress2Go)
Texto Puro (.txt)	1	889 bytes	580 bytes	580 bytes	**34.76%**	**34.76%**
	2	907 bytes	601 bytes	668 bytes	**33.74%**	26.35%
	3	1945 bytes	1249 bytes	1157 bytes	35.78%	**40.51%**
Código-Fonte (.cpp)	1	5345 bytes	3174 bytes	1792 bytes	40.62%	**66.47%**
	2	379 bytes	234 bytes	380 bytes	**38.26%**	-0.26% (expansão)
	3	299 bytes	178 bytes	360 bytes	**40.47%**	-20.40% (expansão)

Desempenho em Arquivos de Código-Fonte:

O ponto de maior sucesso do programa foi sua performance em arquivos de código C++ de pequeno porte (Testes 2 e 3). Nesses casos, a compressão foi significativamente superior à do Compress2Go, que chegou a expandir os arquivos (taxa de compressão negativa). Isso ocorre porque a tabela de frequência fixa do nosso programa é altamente otimizada para a sintaxe do C++. Símbolos como {, }, ;, int e return possuem códigos de Huffman muito curtos. Algoritmos como o Deflate (usado no ZIP) possuem um "custo" inicial (overhead) para construir seu dicionário dinâmico, o que é ineficiente para arquivos muito pequenos.

No entanto, no arquivo de código maior (Teste 1), o Compress2Go foi drasticamente superior. Isso demonstra a principal vantagem dos algoritmos dinâmicos: a adaptabilidade. A ferramenta analisou aquele arquivo específico e criou um modelo de compressão "sob medida", identificando padrões e repetições que a nossa tabela genérica não poderia prever. A nossa tabela é uma ótima média, mas nunca será perfeita para um arquivo individual.

Desempenho em Arquivos de Texto Puro:

Nos arquivos de texto, o desempenho foi competitivo, mas sem uma vantagem clara. No Teste 2, nosso programa foi superior, indicando que a frequência de caracteres em código C++ não é tão diferente da frequência em texto comum. Contudo, no Teste 3, o Compress2Go demonstrou novamente sua capacidade de se adaptar e encontrar o melhor modelo para aquele texto específico, alcançando uma taxa de compressão maior.